Auto-Scaling Heterogeneous Neural Processing Units for Energy and Cost-Efficient LLM Serving

Yuqi Xue
yuqixue2@illinois.edu
University of Illinois Urbana-Champaign

Jichuan Chang
jichuan@google.com
Google

Jian Huang
jianh@illinois.edu
University of Illinois Urbana-Champaign

Abstract

To meet the ever-increasing computing demands of large language model (LLM) services, modern cloud platforms have widely deployed neural processing units (NPUs). These NPU chips have been developed and evolved at an incredibly fast pace, this inevitably produces heterogeneous compute pools backed by different versions of NPU chips. Unfortunately, due to the lack of system and architecture support for managing NPU heterogeneity in the cloud, it is unclear how to best utilize heterogeneous NPUs to maximize the energy and cost efficiency for LLM services.

In this paper, we first conduct a characterization study of various generations of real NPU chips to demonstrate the potential benefits on energy/cost efficiency and performance by utilizing heterogeneous NPU chips. To realize these benefits, we present NeuScale, an auto-scaling framework to automatically exploit heterogeneous NPUs for cloud platforms. NeuScale manages heterogeneous NPU resources with a new vPod abstraction, which abstracts the core hardware parameters of different NPU versions and provides compatibility with existing ML frameworks. It makes the best-fit vPod allocations for different LLM inference requests using an intuitive and lightweight roofline-based analysis. It supports fine-grained dynamic NPU resource provisioning by adjusting both the vPod configuration (i.e., scaling up/down) and the number of vPods (e.g., scaling in/out). To validate the benefits of NeuScale at scale, we implement it with a production-level NPU simulator. Our evaluation with popular LLMs shows that NeuScale can significantly improve cost efficiency and service-level objective (SLO) satisfaction rate by best utilizing heterogeneous NPU resources.

1 Introduction

Cloud platforms today have employed hardware accelerators like neural processing units (NPUs) to support large language model (LLM) services [9, 21, 34, 91]. These NPUs have evolved at a fast pace to meet the computing demands. For instance, Google has produced seven generations of TPUs (a typical example of NPUs) in the past six years [69]. Although different NPU versions share similar architectures (Figure 1), they have different compute capabilities and performance behaviors (Table 2). Their rapid development inevitably creates heterogeneous compute pools in data centers.

Opportunities with heterogeneous NPUs. As the NPU cluster have inevitably become heterogeneous, we need to ensure that different versions of NPUs can be managed and utilized efficiently. Our characterization study of different versions of real NPU chips (see §2.4) shows that no single NPU version is optimal for all workloads. The "best-fit" NPU version depends on whether the LLM workload is compute- or memory-bound, as well as the cost metric we optimize for (e.g., energy or monetary cost). By mapping LLM services to their best-fit NPU resources, we can significantly improve the energy/cost efficiency. Furthermore, as new NPU generations are in high demand, it is often feasible to use older NPUs to meet the performance goals. Reusing old NPUs can not only address the accelerator resource shortage but also amortize their embodied carbon and benefit datacenter sustainability.

Challenges with NPU heterogeneity. Modern cloud platforms lack systematic support for efficiently utilizing heterogeneous NPU chips. This is for three major reasons.

First, cloud platforms lack system software support for heterogeneous NPUs. Although modern cloud vendors such as Google TPU Cloud [69] support NPUs in their compute instances or virtual machines (VMs), they require end users to explicitly specify the NPU version. Given that different NPU versions have diverse computing capability (Table 2) and cost efficiency (§2.4), it is difficult for end users to decide which NPU version fits their workloads the best, and how much NPU resource they should allocate.

Second, given the diverse performance and cost efficiency characteristics of heterogeneous NPU chips, it is inherently challenging to determine the "best-fit" NPU allocation for an LLM workload that satisfies the performance requirement (i.e., service-level objectives, or SLOs) while maximizing energy/cost efficiency. Each LLM model and each inference request may prefer a different NPU allocation (e.g., number of chips, NPU pod topology, model sharding, and batch size), and searching for the best allocation involves complex trade-offs between performance and energy/cost.

Third, cloud platforms lack runtime scheduling support for heterogeneous NPU chips. For LLM services, the workload demand often varies over time (see Figure 17). While cloud platforms have developed auto-scaling techniques [25, 66, 70] to enable automated resource configurations, they were designed with the assumption that managed computing resources are homogeneous, none of them can directly work for heterogeneous NPUs (see §2.5). A new auto-scaling mechanism is highly desirable for heterogeneous NPUs.

Most recently, researchers have been working on the management of heterogeneous GPU clusters [39, 51, 55, 84, 92]. They utilized system profiling techniques to build performance models and learn the optimal GPU configurations (e.g., GPU version and clock frequency) for a specific LLM service. Due to the fundamental architectural differences between NPU and CPU/GPU, the hardware abstraction, performance/energy characteristics, and resource allocation/scheduling mechanisms are all different for NPUs. Managing heterogeneous NPU chips requires new systematic support for resource allocation and runtime scheduling (see §2 and Table 1).

Our solution. In this paper, we present NeuScale, an auto-scaling framework for managing heterogeneous NPUs for LLM services.

Table 1: Comparison of NeuScale with prior works on heterogeneous cluster scheduling and auto-scaling. *Hardware Abstraction*: the structure of allocated hardware resources, which serves as the unit for scheduling. *LLM Sharding-Aware*: the allocation mechanism considers LLM parallelism configurations (e.g., tensor/pipeline/expert parallelisms). *Seq. Len.-Driven*: the scheduling considers LLM request sequence lengths. *Instance Coalescing*: allocated instances will be dynamically merged (see §3.4). *Fail-Over*: when the requested hardware type cannot be allocated, another type (e.g., a different GPU/NPU version) can be allocated. ✓: supported; △: partially supported; ×: not supported.

	Target Hardware	Target Workload	Allocation Mechanism			Scheduling Mechanism				
			Hardware Abstraction	LLM Sharding-Aware	Algorithm	Seq. Len.-Driven	Instance Coalescing	Fail-Over	Scale-Out/In	Scale-Up/Down
Fernandez et al. [17]	CPU	Generic	CPU VM	×	Profiling-based	×	×	×	✓	×
iBalloon [68]	CPU	Generic	CPU VM	×	Profiling-based	×	×	×	×	✓
Autopilot [70]	CPU	Generic	CPU VM	×	Heuristic-based	×	×	×	✓	✓
FIRM [64], AWARE [66]	CPU	Generic	CPU VM	×	Profiling-based	×	×	×	✓	✓
Kubernetes [83]	CPU/GPU	Generic	CPU/GPU VM	×	Heuristic-based	×	×	×	✓	CPU-only
GKE [24, 25]	CPU/GPU/TPU	Generic	VM, container, TPU instance	×	Heuristic-based	×	×	×	✓	CPU-only
Optimus [62]	CPU/GPU	ML Training	N/A	×	Profiling-based	×	×	×	✓	×
Gavel [57]	GPU	ML Training	N/A	×	Profiling-based	×	×	×	×	×
Heet [56]	GPU	ML Training	N/A	×	Profiling-based	×	×	×	✓	×
JABAS [95]	GPU	ML Training	N/A	×	Profiling-based	×	×	×	✓	×
Sailor [77]	GPU	ML Training	GPU VM	✓	Profiling-based	×	×	✓	✓	✓
HAS-GPU [32]	GPU	ML Serving	N/A	×	Profiling-based	×	×	×	✓	Intra-GPU
Splitwise [61]	GPU	LLM Serving	N/A	×	Profiling-based	×	×	×	△	×
ServerlessLLM [18]	GPU	LLM Serving	N/A	×	Heuristic-based	×	×	×	✓	×
SpotServe [52]	GPU	LLM Serving	GPU VM	△	Profiling-based	×	×	△	✓	✓
BlitzScale [96]	GPU	LLM Serving	GPU VM	×	Heuristic-based	×	×	×	✓	×
Mélange [31]	GPU	LLM Serving	N/A	×	Profiling-based	✓	×	×	×	×
DynamoLLM [76]	GPU	LLM Serving	GPU VM	✓	Profiling-based	△	✓	×	✓	✓
NeuScale	**NPU***	**LLM Serving vPod**		✓	**Roofline-based**	✓	✓	✓	✓	✓

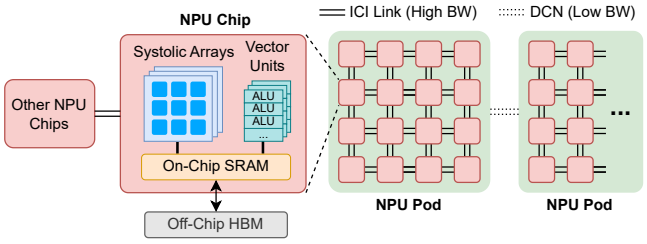


Figure 1: The NPU pod architecture. We draw 2D NPU pods as examples. A pod can be 2D or 3D. ICI: Inter-Chip Interconnect. DCN: Data Center Network.

To ease the management of heterogeneous NPUs, we first develop an abstraction named vPod ("virtualized" NPU pod). It represents a group of NPU chips interconnected with high-speed links. As different NPU versions usually share the same hardware architecture, vPod abstracts core hardware parameters (see Figure 6) and provides a unified interface for different compute capabilities. To simplify vPod management and provide compatibility with existing ML frameworks, each vPod is mapped to the same NPU version. NeuScale exploits NPU heterogeneity by creating multiple vPods.

With vPod abstraction, we develop an NPU allocation mechanism that automatically learns the most energy/cost-efficient vPod allocations, while satisfying the SLO requirement of LLM services. NeuScale utilizes the ML compiler to calculate the arithmetic intensity (FLOPs per byte accessed from HBM) of LLM requests. Based on this, we develop lightweight roofline models to analyze the performance and energy/cost efficiency of each possible vPod allocation. NeuScale identifies Pareto-optimal vPod allocations that represent the best trade-offs between request latency and energy/cost efficiency. We validate that NeuScale can accurately identify best-fit vPod allocations with reasonable analysis overhead (§3.3).

NeuScale employs a closed-loop controller to actuate the best-fit allocations. It auto-scales the vPod allocations to precisely match workload demands at runtime. LLM requests have diverse sequence lengths (see Figure 17), each of which can have a different best-fit vPod allocation. NeuScale creates heterogeneous vPods to accommodate diverse sequence lengths. vPods of the same configuration are organized into groups. As the request rate varies over time, NeuScale dynamically resizes each vPod group and coalesces underloaded groups. When a best-fit vPod cannot be allocated (e.g., the requested NPU version is unavailable), NeuScale will fail over to another NPU version with minimal SLO degradation.

We implement the auto-scaling framework of NeuScale with an NPU cluster simulator, as we cannot access large-scale real NPU chips with different hardware configurations. The simulator models vPod allocation, vPod creation overhead, request scheduling, and common LLM inference engine optimizations [45]. It invokes a production-level NPU simulator as the backend to simulate the execution of each batch of requests on an NPU pod. We validate our simulator with a real-system NPU cluster prototype developed upon different generations of Google TPUs. To the best of our knowledge, this is the first NPU cluster simulator that supports different generations of NPU chips. We will open-source it.

Table 2: Specifications of different NPU chip versions. The data for NPU-A/B/C/D are derived from TPUv4/5e/5p/6e. "*" means the parameter is not officially disclosed, and it is inferred from public data and our experiments.

	NPU-A	NPU-B	NPU-C	NPU-D
Release Year	2021	2023	2023	2024
Process Node	7nm	7nm*	7nm*	4nm*
TFLOPS (bf16)	275	197	459	918
Max. TFLOP/Joule (bf16)	1.07*	0.72*	1.16*	3.50*
Systolic Array Width	128	128	128	256
SRAM Size (MB)	128	64*	128*	128*
HBM Bandwidth (GB/s)	1200	819	2765	1640
HBM Size (GB)	32	16	95	32
HBM Type	HBM2	HBM2E	HBM2E	HBM2E*
Max. HBM GB/Joule	11.06*	6.94*	18.51*	13.52*
ICI BW/link (GB/s)	50	50	100	112
# of ICI links/chip	6	4	6	4
ICI Torus Topology	3D	2D	3D	2D
DCN BW/chip (Gbps)	5	50	50	100
Max. # of chips/pod	4096	256	6144	256

Our evaluation using popular LLMs and service workload traces shows that NeuScale improves energy/cost efficiency by 1.13×/1.43× and SLO satisfaction rate by 1.36× over the state-of-the-art (SOTA) NPU auto-scaling method. As NeuScale offers an efficient way to utilize heterogeneous NPUs, it facilitates NPU reuse and improves datacenter sustainability. Our contributions are as follows:

- We conduct a thorough study to quantify the cost efficiency and performance benefits of exploiting NPU heterogeneity.
- We present NeuScale, an auto-scaling framework to exploit heterogeneous NPUs for energy and cost-efficient LLM serving.
- We propose a new vPod abstraction for managing the allocation and scheduling of heterogeneous NPU chips.
- We develop a new NPU allocation scheme that employs a lightweight roofline-based analysis model to identify the best-fit NPU allocation based on LLM serving workload demands.
- We develop a runtime NPU auto-scaler to dynamically adjust the heterogeneous NPU resource allocation to precisely match changing workload demands.
- We build an NPU cluster simulator that supports different generations of NPU chips. Its codebase will be public on GitHub.
- We evaluate the performance and efficiency of NeuScale with diverse LLM models and production-level traces.

2 Background and Motivation

2.1 Large Language Model Services in the Cloud

Large language model (LLM) services are pervasive in cloud [8, 19, 30, 58, 90]. Serving an LLM request involves two stages [86, 99]: the *prefill* stage processes input tokens in parallel to generate the first output token; the *decode* stage outputs subsequent tokens iteratively. The service-level objectives (SLOs) for LLM serving are defined by two key metrics: the prefill latency (time-to-first-token or TTFT) and the decode time-per-output-token (TPOT).

Due to their substantial computational and memory requirements, LLMs are typically deployed across multiple NPU chips using different parallelism strategies to partition model parameters and computation [46, 47, 74]. The parallelism configuration is selected based on the request pattern (e.g., input/output sequence lengths) and the hardware capabilities. The chosen parallelism configuration affects both the serving performance and cost efficiency.

2.2 System Architecture of NPUs

NPU hardware architecture. Neural processing units (NPUs) are specialized ML accelerators. A typical example is Google TPU [40], the SOTA NPU architecture deployed in production. As shown in Figure 1, an NPU chip consists of specialized compute units for ML computations, including systolic arrays for matrix multiplications and SIMD vector units for generic vector operations. Each NPU chip has an off-chip high-bandwidth memory (HBM) to store the ML model weights and input/output data, and an on-chip SRAM to exploit data locality and hide HBM access latency.

NPU chips can be interconnected via high-speed inter-chip interconnect (ICI) links to form an *NPU pod* (Figure 1). A pod is typically arranged as a 2D/3D torus [100]. Each chip can perform remote direct memory access (RDMA) to another chip's memory. Multiple pods communicate via the traditional data center network (DCN), such as InfiniBand and Ethernet, at a much slower speed than ICI.

NPU software stack. To run an ML workload on an NPU pod, the user defines the ML model using ML frameworks like JAX [81] or PyTorch [7]. The model is represented as a computation graph that specifies tensor operators and the dependencies between operators. The ML compiler [22] determines the parallelism strategies and generates the per-chip graph. Then, it performs optimizations such as operator fusion and tiling, and generates the NPU binary.

2.3 NPU Heterogeneity in the Cloud

In modern cloud data centers, the heterogeneity of NPU chips is becoming common. This is for two major reasons.

First, NPU chips have evolved rapidly to accommodate the ever-increasing demand of ML workloads. Taking Google TPUs as an example (Table 2), the computation power, measured in tera-floating point operations per second (TFLOPS), increased by 3.34× over 3 years; the HBM bandwidth increased by 2.3×; the HBM capacity increased by 3×; and the ICI bandwidth increased by 2×. As the old NPU chips will not immediately retire upon the arrival of new chips, cloud platforms inevitably accumulate multiple NPU generations.

Second, within a single generation, multiple NPU chip variants are designed to optimize for different demands. For example, NPU-B and NPU-D in Table 2 are designed to be "efficiency" chips. They feature a smaller HBM capacity and bandwidth with a high compute-to-memory ratio. In general, they are a good fit for compute-intensive workloads. NPU-A and NPU-C are "performance" chips that have a larger HBM capacity and bandwidth as well as massive FLOPS. In general, they are more optimized for memory-intensive workloads.

2.4 Opportunities of NPU Heterogeneity

We quantify the potential benefits of exploiting NPU heterogeneity for LLM serving. We investigate both dense models (e.g., Llama3 [49]) and mixture-of-experts (MoE) models (e.g., DeepSeek [13]). We employ the SOTA prefill/decode disaggregation [61, 99], as the two phases have distinct demands. We set the latency SLO to be 5× the single request latency on the minimum number of NPU-C or NPU-D chips, whichever is faster [76] (denoted as "5× SLO"). We

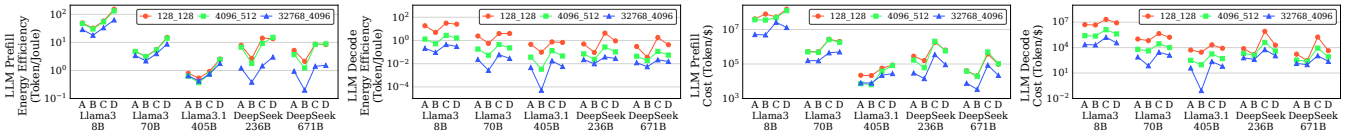


Figure 2: Energy and economic cost efficiency of different NPU versions. The legend labels are `input_output` sequence lengths. The cost is calculated based on $/chip-hour prices of Google TPUs [29].

study both energy and monetary cost. We sweep all SLO-compliant configurations (number of chips, batch size, and parallelism configuration) with a production-level NPU simulator that has been validated against real TPUs (§4), and report the most efficient one. If an NPU version cannot meet the target SLO, we use the configuration with the best attainable latency.

Energy efficiency benefits. Energy efficiency is important for cloud providers, as electricity bills and power provisioning strongly correlate with the total cost of ownership (TCO) and operational carbon emission [41, 72]. Figure 2 shows that the best-fit NPU version achieves 3.5–7.9× higher energy efficiency than the worst-fit for prefill and 3.8–339.7× for decode. This is determined by both the workload's resource demands (e.g., FLOPs/sec, HBM bandwidth, and ICI bandwidth) and the NPU chip's per-component energy efficiency (affected by varying factors like process node, microarchitecture, and circuit-level optimizations). LLM prefill is compute-bound, so the energy efficiency is determined by the computational energy efficiency (TFLOP/Joule). Hence, NPU-D is the most energy-efficient (see Table 2). LLM decode is memory bandwidth-bound, so NPU-C is the most energy-efficient due to its high HBM GB/Joule.

Monetary cost savings. In Figure 2, we also quantify monetary cost [29] (token/dollar), a common optimization goal for third-party users who rent NPU instances and deploy their own LLM inference services in the cloud. The monetary cost of the best- vs. worst-fit NPUs can differ by 3.3–34.2× for prefill and 5.5–2,602× for decode. Monetary cost is tied to a flat chip-hour price, making it preferable to select the NPU version that minimizes total chip-hours.

> **Takeaway (T1):** The best-fit NPU version depends on an ML workload's bottlenecking resource (e.g., compute, memory) and the specific cost metric (e.g., energy vs. monetary cost). Exploiting heterogeneity can satisfy diverse optimization goals.

An LLM service needs to serve requests with diverse sequence lengths (see Figure 17). Each request can have a distinct resource bottleneck. A one-size-fits-all NPU pod configuration is inherently sub-optimal. In Figure 3, the most energy-efficient configuration (defined by the NPU version, pod shape, parallelism strategies, and batch size) varies significantly across various sequence lengths. Exploiting NPU heterogeneity at fine granularity enables us to optimize efficiency for each request with its "best-fit" NPU allocation.

> **Takeaway (T2):** Different LLM inference requests can favor different NPU versions and NPU pod configurations. It is desirable to provision a mix of heterogeneous NPU pods at fine granularity to satisfy diverse resource demands.

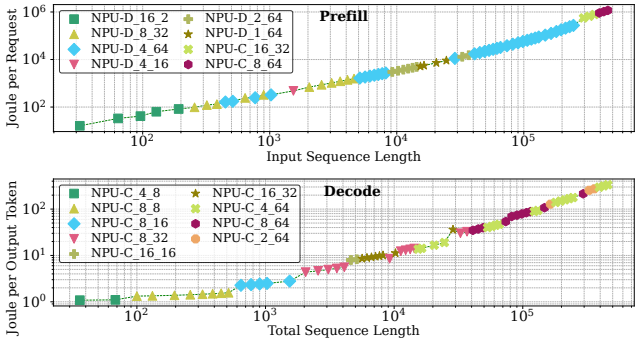


Figure 3: The optimal energy efficiency of LLM requests with various sequence lengths. The legends "{NPU type}_{tensor parallelism degree}_{pipeline parallelism degree}" indicate the optimal configuration for the specific sequence length. Due to space limitation, we only show Llama3.1-405B here.

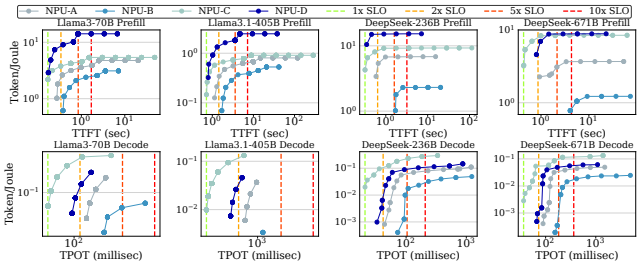


Figure 4: Optimal energy efficiency vs. latency across NPU versions. Monetary costs and Llama3-8B have similar trends (omitted due to space limitations).

Performance fungibility. As new chips are scarce, it is not always possible to allocate the best-fit NPU version. Therefore, it is often possible for an alternative NPU version to meet the performance goals. Such *performance fungibility* helps alleviate the shortage of the latest NPUs. As shown in Figure 4, there is always at least one NPU version that can satisfy ≥ 2× SLO. In practice, a strict 1× SLO is rare, as it forbids batching even on the most powerful NPUs [93].

> **Takeaway (T3):** It is often feasible to use older NPU versions without compromising service quality. This helps mitigate supply shortages of new NPUs and improve resource availability.

2.5 Challenges with Heterogeneous NPUs

Despite the performance and efficiency benefits, exploiting NPU heterogeneity is challenging due to three major reasons.

First, cloud platforms today lack system software support for heterogeneous NPUs. They manage different NPU versions as separate, homogeneous resource pools, and rely on end users to specify the NPU version [21, 87]. Given the diverse computing capabilities and cost efficiency, it is cumbersome for users to decide which NPU version and how many NPU chips to allocate.

Second, given the diverse characteristics of heterogeneous NPUs, it is challenging to find the "best-fit" NPU allocation due to the large search space defined by parameters including NPU version, pod shape, model sharding, and batch size. While we can employ analytical models to predict LLM inference latency [97], they typically model execution at the operator level and cannot capture the impact of different software stack optimizations. Moreover, due to differences in hardware architecture and computing stacks, it is hard to apply them directly to heterogeneous NPUs.

Third, cloud platforms lack runtime resource management for heterogeneous NPUs. For production LLM services, the workload demand varies over time (see Figure 17). They are commonly deployed with auto-scaling to dynamically adjust the allocated resources [25, 28, 66, 70]. However, current auto-scaling techniques assume the underlying NPU pool is homogeneous and only adjust the number of allocated NPU pods (i.e., scale-in/out). A potential solution is to also adjust the configuration of each NPU pod (i.e., scale-up/down). While there have been scale-up/down techniques for CPU/GPU VMs [66, 70], none of them work for heterogeneous NPUs due to fundamental architectural differences.

The unique properties of NPU architecture require careful design of the corresponding system stack for enabling the efficient use of heterogeneous NPU chips in LLM serving. As shown in Table 1, an efficient auto-scaling system for LLM serving requires a systematic support, which includes appropriate hardware abstraction, allocation mechanism, parallelism support of LLMs, scheduling mechanism, exception (fail-over) handling, and others. Unfortunately, NeuScale cannot simply borrow or reuse existing auto-scaling design and implementation for CPU/GPU architectures, due to the special requirements/consideration for hardware abstraction, performance/energy trade-offs, resource allocation and scheduling mechanisms in heterogeneous NPU management. We will discuss how we overcome each of the challenges throughout §3.

3 Design and Implementation

We design NeuScale, a fine-grained auto-scaling framework for heterogeneous NPUs to achieve the following goals: (1) **Efficiency:** NeuScale should map LLM requests to their best-fit NPU pods to maximize energy/cost efficiency; (2) **Performance:** NeuScale aims to meet SLOs of LLM requests; (3) **Flexibility:** When the best-fit NPUs are not available, NeuScale should automatically fail over to another NPU version with minimal SLO and efficiency degradation; (4) **Compatibility:** NeuScale should be compatible with existing ML software stack and cloud infrastructure.

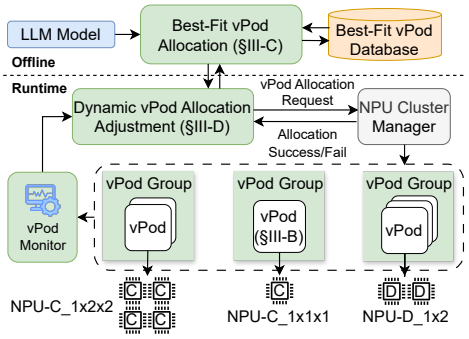


Figure 5: System architecture of NeuScale. `NPU-C_1x2x2` is an 1x2x2 NPU-C pod. `NPU-C_1x1` is a single NPU-C chip. `NPU-D_1x2` represents two interconnected NPU-D chips.

```
struct vChipConfig {
    int flops_per_sec;      // peak FLOPS
    int hbm_bw_GBps;        // peak HBM bandwidth
    int ici_bw_GBps;        // peak ICI bandwidth
    int HBM_capacity_GB;    // per-chip HBM capacity
    int sram_capacity_MB;   // per-chip SRAM capacity
};
struct vPodConfig {
    NPUVersion version;     // mapped physical NPU chip version
    vChipConfig chip_cfg;   // NPU chip config
    int num_chips;          // number of chips
    ICITopology topology;   // 2DTorus or 3DTorus
    int pod_shape[3];       // shape of the 2D/3D torus
};
```

Figure 6: vPod configuration parameters.

3.1 NeuScale Overview

Figure 5 shows the system architecture of NeuScale. To simplify the management of heterogeneous NPUs, we introduce a new vPod abstraction in §3.2. We describe how NeuScale learns workload characteristics with ML compilers to derive the "best-fit" vPod allocation that can be shared across all workloads using the same backbone LLM model in §3.3. Then we show how NeuScale auto-scales the vPod allocations to match the workload demand at runtime in §3.4.

3.2 vPod Abstraction for Heterogeneous NPUs

The vPod abstraction serves three purposes: (1) It provides a unified representation for different NPU pod configurations. (2) It abstracts out the parameters needed for NeuScale to allocate and schedule NPU resources. (3) It provides compatibility with the ML workloads and existing cloud infrastructure.

vPod abstraction. A vPod represents a group of NPU chips interconnected via high-speed ICI links. Figure 6 shows its configurable parameters. At the pod level, a vPod specifies the chip count and ICI topology. At the chip level (`vChipConfig`), it contains the on/off-chip memory capacities and key performance parameters, including peak FLOPS, HBM bandwidth, and ICI bandwidth. The `version` field tracks which physical NPU chip version this vPod is mapped to, allowing ML frameworks to perform device-specific optimizations.

All NPU chips in a single vPod are homogeneous. This aligns with the organization of physical NPU pods, as different NPU versions have incompatible ICI links and will not be mixed in the same

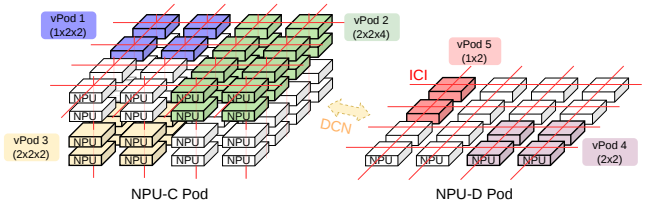


Figure 7: Mapping of vPods to different slices of physical NPU pods. NPU-C supports 3D torus, and we draw a 4x4x4 slice as an example. An NPU-C pod can scale to 16x16x24. NPU-D supports 2D torus, and the largest pod is 16x16. Each NPU chip is connected to its neighbors via ICI links. Different NPU versions have incompatible ICI links, so they are in different NPU pods and must communicate over DCN.

physical pod. This also offers the best compatibility with existing ML frameworks and simplifies model deployment.

vPod deployment. To create a vPod, NeuScale submits vPod allocation requests to the NPU cluster manager (e.g., Borg Scheduler [87] or Kubernetes kube-scheduler [82]), which finds the available NPU resources to map this vPod. To minimize the chance of fragmentation, NeuScale restricts the vPod topology to powers of two in each dimension, which aligns with the underlying physical NPU pods [40, 100], as shown in Figure 7. Each vPod is mapped to a dedicated mesh of NPU chips, which provides physical isolation for compute, memory, and ICI. After vPod creation, NeuScale initializes LLM inference engine (e.g., vLLM [45], JetStream [20]).

3.3 Best-Fit vPod Allocation

To allocate a vPod for an LLM model, we need to determine (1) the vPod configuration, (2) the LLM parallelism configuration, and (3) the maximum batch size. These parameters are interdependent: the vPod configuration will limit how the model can be partitioned; parallelism configuration and batch size determine the computation task on each chip, which significantly affects performance and energy/cost efficiency. We must co-optimize all factors.

Our key insight is that the energy/cost efficiency is determined by a workload's bottlenecking resource **(T1)**, and we can identify it by learning the workload's arithmetic intensity (i.e., FLOPS per byte accessed from HBM). Based on this, we can use a lightweight roofline model [89] to predict the energy/cost efficiency and the execution time of an allocation. With these predictions, we find all Pareto-optimal allocations that represent the optimal trade-offs between execution time and energy/cost efficiency, and pick the most efficient allocation that satisfies SLO.

Learning the arithmetic intensity with ML compiler. The arithmetic intensity depends on the model architecture, parallelism strategies, sequence length, batch size, and compiler optimizations (e.g., operator fusion and tiling). To systematically cover these factors, we extend the XLA compiler [22, 60] with three analysis passes to perform a fast design space exploration (DSE).

First, the *vPod configuration generation pass* enumerates all candidate vPod configurations (i.e., the NPU chip version and the pod shape). To reduce the search space size, we limit the number of chips per pod to 1024 (sufficient for a single LLM model replica).

Second, for each vPod configuration, the *parallelism configuration pass* determines all valid combinations of parallelism degrees (e.g., tensor/pipeline/expert parallelisms). For each combination, it also generates per-chip computation graphs for batch sizes that are power-of-2 ($1 \le bs < 16$) or multiple-of-16 ($16 \le bs \le 1024$) [45] by default. These batch size buckets are sufficient to cover most request patterns and are user-configurable. For each graph, we apply standard graph- and operator-level optimizations (e.g., operator fusion, tiling) to ensure an accurate estimate of the arithmetic intensity.

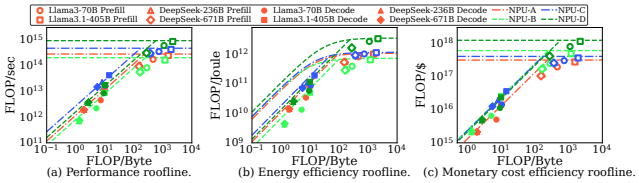
Finally, the *arithmetic intensity analysis pass* invokes high-level operation (HLO) cost analysis [59] on each operator to extract its FLOP count and HBM traffic. The extracted data can be fed into the roofline model for performance, energy, and cost predictions.

Predicting performance and efficiency with roofline model. We rethink roofline models as tools for quantifying work-done-per-cost, where "work done" is total FLOPs and "cost" is an arbitrary metric, such as time, energy, or monetary price. We use a roofline model, because it is lightweight and can accurately capture the performance trend based on the hardware specifications and model architecture with low overhead (see Figure 9 and "Allocation time overhead" at the end of §3.3). We do not use profiling-based approaches [31, 76], as they incur high performance overhead, due to the need to profile a large number of vPod configurations (every combination of batch size, sequence length, LLM parallelism configuration, and NPU pod shape needs to be profiled).

Figure 8 shows the roofline models for different cost metrics. An NPU chip is represented by two rooflines: the horizontal line is the peak computational efficiency (FLOP/cost); the sloped/curved line is the peak HBM bandwidth efficiency (byte/cost). These rooflines can be derived from the hardware specifications provided by the vendor (e.g., peak FLOPs, memory bandwidth, and energy efficiency) and the instance pricing from the cloud platform. The energy efficiency roofline (Figure 8b) is a curve instead of a straight line. This is because for each FLOP, we need to fetch data from HBM, so the peak FLOP/Joule also depends on HBM accesses. The monetary cost roofline (Figure 8c) is based on dollars per chip-hour and hence proportional to the performance roofline (Figure 8a).

For each candidate vPod allocation generated by the ML compiler, we can plot it as a vertical line at its arithmetic intensity. The intersection of this vertical line with the roofline predicts the performance/efficiency. NeuScale uses the performance roofline to predict the request latency, and energy/monetary cost roofline to predict the cost. For the cost metric, first-party cloud platforms may prioritize energy efficiency, as electricity bills are highly correlated to TCO [72]. Third-party users who rent vPods may optimize monetary costs, as instances are billed per chip-hour.

NeuScale takes the performance and cost of all candidate vPod allocations and identifies the Pareto-optimal ones. NeuScale picks the most energy/cost-efficient allocation that satisfies the SLO constraint of the sample input request. To generate the best-fit vPod allocations for all representative sequence lengths, NeuScale performs offline DSE with sample requests grouped into sequence length buckets. This only needs to be performed once, and the results are shared across all workloads that use the same LLM model. The generated best-fit database size is proportional to the number of sequence length buckets and the number of NPU versions. By default, NeuScale creates sequence length buckets with 128-token



(a) Performance roofline. (b) Energy efficiency roofline. (c) Monetary cost efficiency roofline.

Figure 8: Cost-efficiency roofline models with different cost metrics. Marker shapes represent LLM models. Colors represent NPU versions. We only plot input/output sequence length 4K/512 due to space limitations.

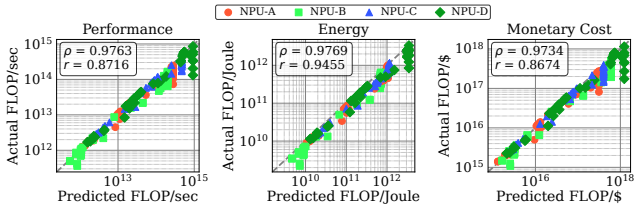


Figure 9: Roofline prediction accuracy for Llama3.1-405B and input/output sequence lengths 4K/512. ρ is Spearman's rank correlation. r is Pearson correlation. Each point represents the predicted/actual value of an allocation.

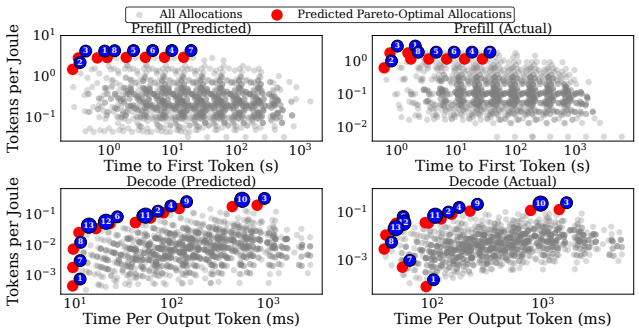


Figure 10: Predicted vs. actual latency and energy efficiency of all allocations. The points with the same label in the predicted and actual graphs correspond to the same allocation. We only plot DeepSeekV3-671B with input/output sequence length 4K/512 due to space limitations.

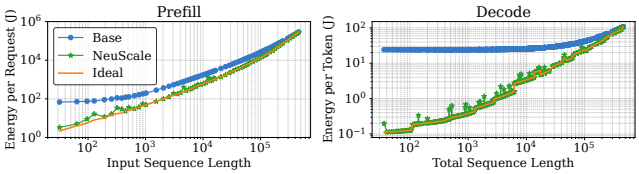


Figure 11: Allocation accuracy (Llama3-70B). `Base` uses the best-fit allocation for the largest sequence length in order to guarantee SLO (see §5.1).

increments up to 1024 tokens. Beyond that, we double the increment at each power-of-2 boundary, e.g., 256-token increments up to 2048 and 512-token increments up to 4096 [45]. With a maximum of 1M sequence length, this yields $48 \times 2 = 96$ (total for prefill and decode) buckets. For each bucket, the database stores a best-fit vPod configuration (i.e., all fields in Figure 6, plus metadata such as the maximum batch size, LLM parallelism degrees, and the estimated throughput/latency, taking up to 128 bytes) for each NPU version. With 4 NPU versions, the database is only $96 \times 4 \times 128\text{B} = 48\text{KB}$ per model. At runtime, the database can be easily cached in memory as a hash table, and the query latency is negligible ($\leq 20\mu$s).

Roofline prediction accuracy. Figure 9 shows the accuracy of our roofline models. The Spearman correlation ρ>0.97 ($\rho = 1$ means predictions preserve all relative ranking), despite a bit lower Pearson correlation r ($r = 1$ implies a perfect linear relationship). This implies that if an allocation is predicted to be better than another, it is highly likely to be actually better, which is sufficient for DSE.

NeuScale allocation accuracy. Figure 10 shows the predicted Pareto-optimal allocations (left) and their actual latencies and energy efficiencies (right). The predicted Pareto curve covers all actual Pareto-optimal allocations despite some false positives. In practice, testing an allocation on real hardware is necessary to guarantee SLO and is a common safeguard before deployment [27, 85]. As we only need to test a few Pareto-optimal allocations, we can minimize this testing cost. Figure 11 shows the end-to-end energy efficiency for different allocation policies (see the detailed setup in §5.1). NeuScale successfully finds the best-fit allocation for most sequence lengths and achieves near-ideal energy efficiency.

Allocation time overhead. The analysis passes take up to 40 seconds on a Cloud TPU VM (120 AMD EPYC 7B12 CPU cores) for each sharded graph for the largest DeepSeekV3-671B model. In practice, the entire DSE can finish in 10 minutes for each sequence length. This is acceptable, as the DSE can be parallelized across multiple servers, and performed offline. The results are saved in a database and shared by all services that use the same LLM.

3.4 Dynamic vPod Allocation Adjustment

At runtime, NeuScale organizes vPods into *vPod groups*, all vPods in the same group have the same configuration, and each group is the best-fit for specific sequence lengths. The *vPod monitor* tracks the runtime statistics of all vPods in the last time window (30 minutes by default). At every epoch (5 minutes by default), it reports the statistics to the *vPod allocation recommender*, which adjusts the number of vPod groups and the number of vPods in each group. Globally, NeuScale tracks the sequence lengths of all requests in the last time window. For each vPod, NeuScale collects its incoming request rate as a time series of measurements sampled every 10 seconds. Figure 12 shows the end-to-end auto-scaling process.

❶ **Creating a vPod group.** The recommender queries the DSE module (§3.3) to determine the best-fit vPod allocation for each unique sequence length. It has negligible overhead if the allocations are already generated offline. Otherwise, the recommender picks the best-fit allocation for the closest sequence length in the database and triggers the background DSE for the new sequence length. NeuScale compares the new set of vPod allocations with existing vPod groups to determine if a new group needs to be created.

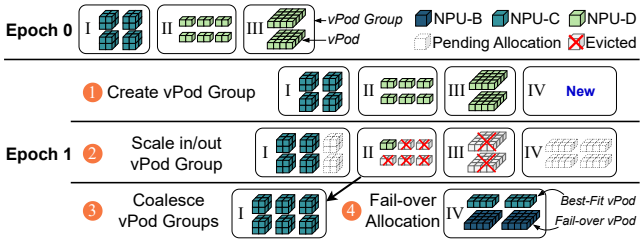


Figure 12: vPod auto-scaling process in NeuScale.

2 Scaling in/out a vPod group. NeuScale determines the vPod count in a group (N_{pod}) based on the peak request rate (R_{peak}) in the last time window that would be served by this group and the maximum throughput of each vPod (T_{pod}): $N_{pod} = \lceil R_{peak}/T_{pod} \rceil$. NeuScale evicts a vPod group if N_{pod} is scaled down to 0. To avoid thrashing, the vPod count can at most double in every epoch and halve every 30 minutes (user-configurable). To handle bursty workloads, NeuScale monitors the peak aggregated queue size of each vPod group. If this size exceeds the group's aggregated maximum batch size, NeuScale triggers an emergency vPod allocation adjustment. This reactive scale-out bypasses the doubling constraint to quickly absorb workload spikes. To minimize underutilization once a burst diminishes, we apply an exponentially weighted moving average to the R_{peak} signal, allowing the system to scale in rapidly.

3 Coalescing underutilized vPod groups. When the sequence lengths are sparse, each request may have a different best-fit vPod allocation, leading to underutilization if a separate vPod group is naively created for each request. To mitigate this, NeuScale merges a vPod group into the group serving the next larger sequence length if the group's normalized peak throughput $N_L = R_{peak}/T_{pod}$ falls below a configurable threshold (0.5 by default), and the target group has sufficient spare capacity to absorb all its requests without allocating a new vPod. NeuScale repeatedly scans groups in ascending order of N_L and performs merges until no further such cases remain.

4 Fail-over vPod allocation. When a best-fit vPod cannot be allocated (e.g., due to resource scarcity, fragmentation, or exceeded quota), NeuScale automatically fails over to the best-fit vPod on another NPU version. The fail-over vPod belongs to the same group as the best-fit ones to simplify request routing and vPod group scaling. In each epoch, NeuScale will replace as many fail-over vPods as possible. If the fail-over vPod allocation also fails (indicating resource scarcity or a quota limit), NeuScale will evict vPods from the group with the lowest R_{peak} to provision the new vPod.

NeuScale request routing. NeuScale employs a BERT-based model to predict the output sequence length based on the input prompt [63, 65, 80]. It routes requests to their best-fit vPod groups with a sequence length-to-vPod group mapping, which is updated when vPod groups are created or evicted. In each group, NeuScale employs optimizations such as load balancing and continuous batching [45]. To handle output length mis-predictions, NeuScale re-estimates the remaining output length of a request whenever its sequence length reaches the next bucket boundary (see §3.3 for the bucket definitions). When either the new estimate or the current sequence length exceeds twice the best-fit sequence length for the current vPod, NeuScale triggers a non-blocking request migration [78] to move the request to the vPod group responsible for longer sequences. The mis-prediction penalty and request migration overhead are tolerable (see our evaluation in §5.6).

Table 3: vPod system-level overhead. The numbers are measured on TPU instances on Google Cloud Platform [21]. *DCN BW varies for different NPU versions (see Table 2).

Overhead Source	Time
Best-fit vPod lookup	≤20 μs
Create a vPod instance	73–108 s
Delete a vPod instance	~15 s
Download sharded model weights	5–100 Gbps/chip (DCN BW*)
NPU runtime & LLM serving engine startup	~31 s
Load model weights to NPU chips	24 GB/s/chip (PCIe 4.0 x16)
Query vPod runtime statistics	2–5 s
vPod allocation adjustment algorithm	~100 ms
Request routing latency	45.9 ms
KV cache transfer between vPods	see Figure 13

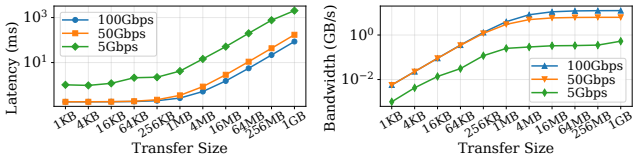


Figure 13: P2P data transfer latency and bandwidth between two chips over DCN, measured on real TPUs. The two chips are in different vPods.

4 Methodology

Real system prototype. To validate our design, we developed a real system prototype and deployed it on a real TPU cluster consisting of 32 TPUv4, 64 TPUv5e, and 64 TPUv6e chips (three on-demand TPU instances: `TPUv4-64`, `TPUv5e-64`, and `TPUv6e-64`[1]). The offline DSE (§3.3) is implemented as a standalone Python module. It first enumerates all vPod configurations, sequence lengths, LLM parallelisms, and batch sizes for a model. Then, it invokes the XLA compiler to generate and compile all execution graphs with extra compiler flags to dump the intermediate HLO cost analysis pass results. Finally, it invokes the roofline-based allocation to generate the best-fit database. The runtime auto-scaling controller (§3.4) lives in a dedicated CPU VM to manage all TPU VMs (a TPU instance has multiple TPU VMs and 4–8 TPU chips per VM). It bookkeeps the vPod-to-TPU chip mappings and queries the runtime statistics from the vLLM [45] engines running in the TPU VMs. To (de)allocate a vPod, it utilizes `ssh` to access the corresponding TPU VMs and launch/kill the vLLM [45] engines. We extended the vLLM router [1] to support sequence length-based request routing and Llumnix-style non-blocking request migration [78].

Simulator. To evaluate NeuScale at scale, we build a simulator to model the auto-scaling framework, including dynamic vPod allocation and inter-vPod request routing. For each vPod, it models common LLM inference optimizations such as continuous batching, KV cache swapping [45], and request migrations across vPods in

[1]We did not have access to TPUv5p instances as they are scarce and in high demand. However, TPUv5p shares the same core architecture as TPUv5e.

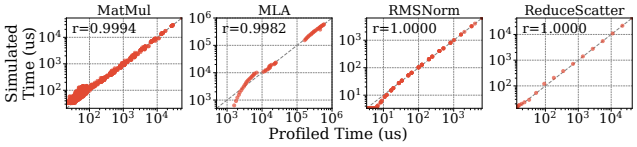


Figure 14: Simulated vs. profiled execution time of representative operators on a $2 \times 4 \times 4$ TPUv4 pod. Each data point corresponds to a different input tensor shape and tensor parallelism degree. r is the Pearson correlation.

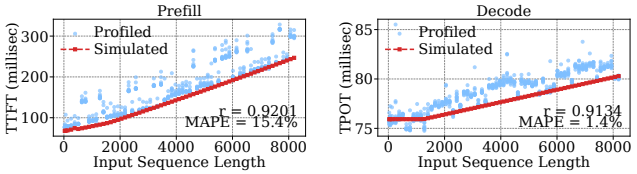


Figure 15: Simulated vs. profiled latency of a single LLM inference request with various sequence lengths. r is Pearson correlation. MAPE is the mean absolute percentage error. We show the results of Llama3-70B on a $2 \times 4 \times 4$ TPUv4 pod.

the same group for load balancing [78]. We model the major system-level overheads for auto-scaling, as shown in Table 3. We calibrated the overhead based on measurements on real TPU instances. We compute model weight downloading and loading time based on the model size, parallelism configuration, and measured network download and PCIe bandwidths of TPU VMs. To model KV cache transfer overhead over DCN due to request migration and prefill/decode vPod communication, we quantify the communication overhead in Figure 13 with different TPU versions (see DCN bandwidth in Table 2) and fit a link model [2, 4] based on the measured numbers.

For each request, a backend production-level NPU chip simulator is invoked to get its latency and energy. The backend takes the per-chip execution graph and simulates all major components on an NPU chip (SA, VU, SRAM, HBM, and ICI). It reports the per-operator statistics for each component, including execution time, FLOPs utilization, and memory/ICI traffic. For MoE models, based on empirical studies[36], we adjust the factor such that the most loaded expert receives 7× as many tokens as an average expert. We model the power of each component and combine the power model with the performance to quantify energy consumption. For SA, VU, and SRAM, we implement them in RTL, synthesize and place & route the components with ASAP7 PDK [12] using Synopsys toolchain to obtain the power estimation. For HBM/ICI, we derive the power based on public TPU data [40, 42, 79, 100], Google's datacenter optical network [75], and HBM IP datasheets [6, 14, 67].

Simulator validation. We verified our NPU chip simulator with different TPU versions. We validated the execution time of all operators in the LLMs studied in the paper. We show representative operators in Figure 14. We validated the TTFT and TPOT of models with different parallelism degrees, sequence lengths, and batch sizes. Figure 15 shows the validation results for Llama3-70B with tensor parallelism degree 32 and batch size 1 as an example. Our simulator

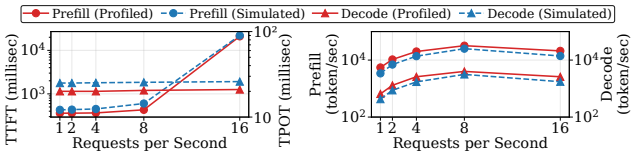


Figure 16: Simulated vs. profiled TTFT, TPOT, and throughput of NeuScale at steady state on a small TPU cluster under different request rates.

Table 4: LLM inference traces used in our evaluation.

	Request Rate (average req/minute)	**Input Seq. Len.** (min/avg/max)	**Output Seq. Len.** (min/avg/max)
Azure [76]	1667	1/2556/7691	1/23/3500
LVEval [94]	41	8.8K/105K/431K	1/17/278
OpenThoughts [33]	27	5/246/12K	531/11.6K/16.8K

achieves high correlations with profiled results. We also validated our power model against the published TPU data [40–42, 79].

We further validate our cluster-level simulator against our real-system prototype serving Llama3-70B, running both with synthetic workloads sampled from the Azure trace [54], sweeping request rates and reporting steady-state performance in Figure 16. Since our simulator achieves high correlation at operator- and request-level, and it implements the same vPod allocation and request routing algorithms as the real-system prototype, it faithfully reproduces the trends observed in the real-system: (1) the prefill throughput saturates at 8 requests/second, and the prefill TTFT increases drastically with a higher request rate; (2) the decode throughput drops at 16 requests/second due to prefill bottlenecks, but the TPOT is not affected since decode instances are not yet saturated.

5 Evaluation

Our evaluation shows that (1) NeuScale improves energy/cost efficiency by 1.13×/1.43× and SLO satisfaction rate by 1.36× over SOTA baselines (§5.2); (2) its benefits generalize to diverse LLM inference workloads (§5.3) and various NPU cluster configurations (§5.4); (3) it still achieves a high SLO satisfaction rate under resource scarcity (§5.4); (4) its fine-grained vPod grouping strategy adapts to sequence length variance more effectively than a coarse-grained multi-pool approach (§5.5); (5) it is robust to output sequence length prediction accuracies (§5.6); (6) On a real heterogeneous TPU cluster, NeuScale improves cost efficiency by up to 11.2% and always maintains high SLO satisfaction rate (§5.7); and (7) it facilitates reusing old NPUs and helps improve datacenter sustainability (§5.8).

5.1 Experimental Setup

Workloads. We evaluate both dense models (Llama) and sparse Mixture-of-Experts (MoE) models (DeepSeek). We replay production-level traces as shown in Table 4. Azure [54] is a production-scale trace for coding services. LVEval [94] is a long-context Q&A dataset. The OpenThoughts dataset [33] contains complex questions in math, coding, and science, and triggers long reasoning outputs. All models use BF16 weights. As the traces exhibit daily periodicity, we

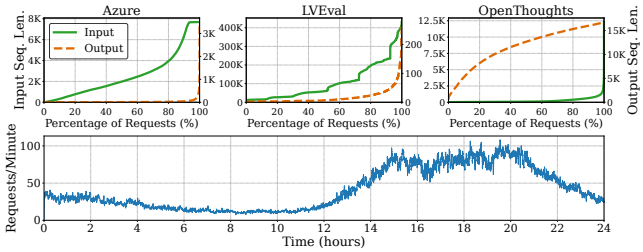
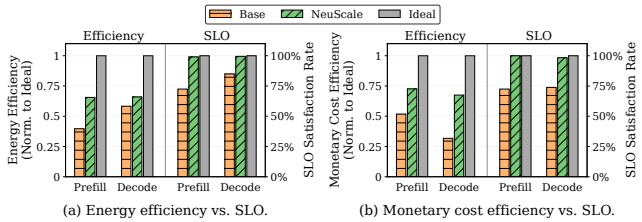


Figure 17: Sequence length and request rate of traces.



(a) Energy efficiency vs. SLO. (b) Monetary cost efficiency vs. SLO.

Figure 18: End-to-end energy/cost efficiency and SLO satisfaction rate on DeepSeekV3-671B (the largest model in our evaluation) with `Azure`. §5.3 covers other workloads.

replay a 24-hour segment (see Figure 17). To better represent workloads with long sequence lengths (e.g., reasoning models and agentic applications), we create synthetic traces by sampling sequence lengths from two recent datasets [33, 94] and issuing requests following the timestamps in the Azure trace. As the Azure trace only contains the timestamp and sequence length of each request, we simulate a 60% output sequence length prediction accuracy by default (i.e., when a request needs to be routed, there is a 60% chance it is routed to the correct vPod group; otherwise, it is routed to a random vPod group whose sequence length is no shorter than the request's current sequence length). We study different accuracies in §5.6. We scale the request rate such that the total tokens per minute is similar to that in the original Azure trace, while preserving the temporal pattern [52, 71, 96] (see Table 4). We set the SLO following §2.4. We deploy separate vPods for prefill/decode phases [99] managed by two independent auto-scalers.

Testbed setup. We configure the simulator for a cluster containing all four NPU versions from Table 2. We treat NPU-C/D as the newer, more powerful generations, NPU-A/B are the older chips. By default, we assume a workload has enough resource quota, and a vPod allocation always succeeds. This is typical for cloud services, as service providers typically choose a conservatively large quota to ensure service quality. We study the impact of insufficient quota in §5.4. We also evaluate our NeuScale prototype on a real heterogeneous TPU cluster in §5.7 (see the cluster setup in §4).

Baselines. We evaluate NeuScale with the following designs.

- *Base*: SOTA auto-scaling method that scales only vPod count and uses homogeneous configuration [70, 83], which is commonly used in modern cloud platforms like Google GKE [25] (see Table 1). It uses the best-fit vPod allocation for the largest sequence length of each trace to provision for the peak demand.
- *NeuScale*: our design for auto-scaling heterogeneous vPods. We generate the best-fit allocations for each unique sequence length for each model before running the experiments.
- *Ideal*: an oracle design that always achieves the best efficiency while meeting SLO for each request. We derive the ideal efficiency for a request by first finding its best-fit vPod allocation. We then run a maximum-sized batch sharing that same sequence length on the best-fit vPod to obtain the ideal energy/cost.

5.2 Efficiency and SLO Satisfaction Rate

We evaluate the end-to-end energy/cost efficiency and SLO satisfaction rate of NeuScale. We focus on DeepSeekV3-671B, the largest model in our experiments, and the `Azure` trace (see §5.3 for sensitivity analysis with other workloads). Figure 18 summarizes the results. Overall, NeuScale can improve the energy efficiency by up to 1.65× (1.37× on average), the monetary cost efficiency by up to 2.13× (1.73× on average), and the SLO satisfaction rate by 1.38× (1.31× on average), as compared to *Base*.

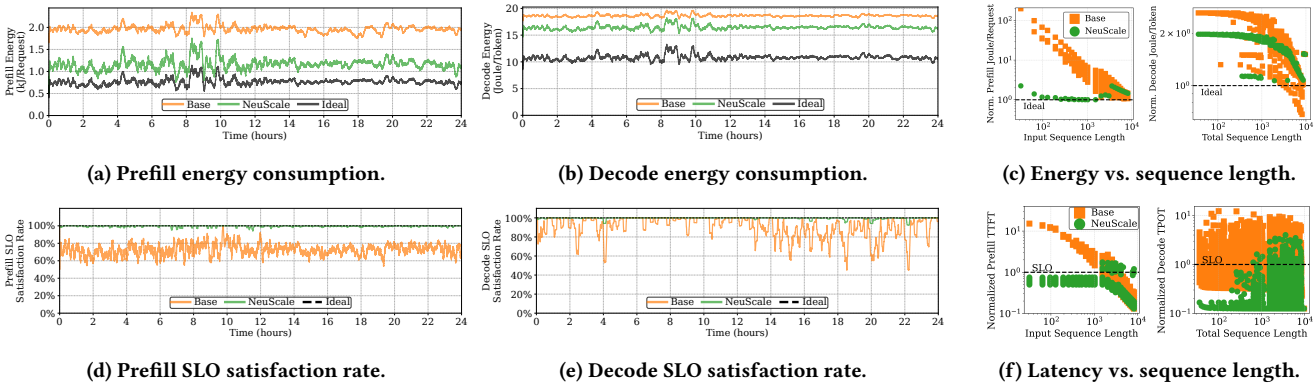
The one-size-fits-all strategy of *Base* suffers from two limitations. First, it leads to energy/cost inefficiency, as the best-fit vPod for the largest sequence length is large and contains many NPUs. Running small requests on them underutilizes their NPUs. Second, large vPods may employ a high tensor parallelism degree to exploit the aggregated FLOPS and memory bandwidth of all NPUs. Running small requests on them may suffer ICI bottleneck, making the latency worse than on a smaller vPod and leading to SLO violations.

NeuScale outperforms *Base* by allocating the best-fit vPod for each sequence length. The gap between NeuScale and *Ideal* is mainly due to reactive auto-scaling at runtime. Since NeuScale adjusts the vPod allocation based on the peak load and sequence lengths observed in the last 30-minute time window, if the actual current load is lower, some vPods may not be fully utilized. If NeuScale cannot find a best-fit vPod instance for a request (e.g., due to unseen sequence lengths or vPod group coalescing), this request may suffer from sub-optimal vPod allocation. Despite these limitations, NeuScale still significantly improves the energy/cost efficiency over *Base*, and can achieve more than 80% of ideal efficiency for 22 out of 48 workloads in our experiments (see §5.3).

Benefits breakdown. As shown in Figure 19, NeuScale consistently delivers higher energy efficiency and SLO satisfaction rate than *Base* regardless of the changes in request load (see Figure 17). The benefits come from mapping each request to their best-fit vPods. As shown in Figures 19c and 19f, NeuScale improves the energy efficiency and latency of most requests. In contrast, *Base* is only optimized for the largest requests, and it suffers severe energy and latency overhead for smaller requests. We visualize how NeuScale auto-scales vPods over time in Figure 20. For prefill, the number of vPods in each group follows the request rate changes. For decode, the groups `D1` and `D2` are larger than others. They are the best-fit vPods for requests with total sequence length greater than 2K, which account for more than half of the requests in `Azure`.

5.3 Sensitivity Analysis with Various Workloads

We evaluate NeuScale with different models and traces in Figure 21 and Figure 22. On average, NeuScale improves the energy efficiency, monetary cost efficiency, and SLO satisfaction rate by 1.13× (up to



(a) Prefill energy consumption. (b) Decode energy consumption. (c) Energy vs. sequence length.

(d) Prefill SLO satisfaction rate. (e) Decode SLO satisfaction rate. (f) Latency vs. sequence length.

Figure 19: Left/middle columns: energy consumption and SLO satisfaction rate over time. Right column: energy/latency of each request w.r.t. its sequence length. We use DeepSeekV3-671B with `Azure` as an example and analyze other workloads in §5.3.

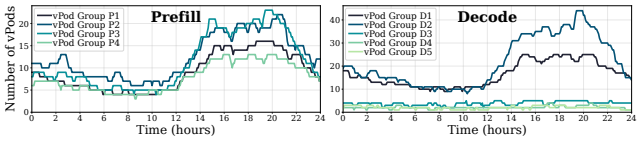


Figure 20: Number of vPods in each vPod group over time. We plot DeepSeekV3-671B the `Azure` trace as an example.

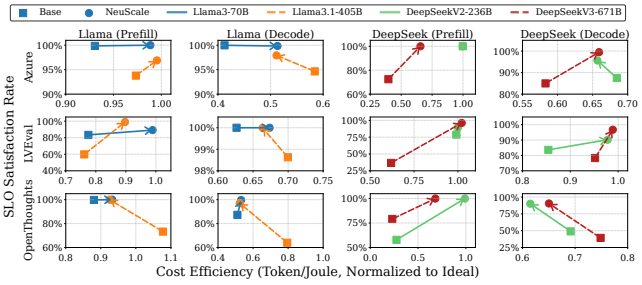


Figure 21: Energy efficiency vs. SLO satisfaction rate. Different colors represent different models. Arrows indicate the improvement from *Base* to NeuScale.

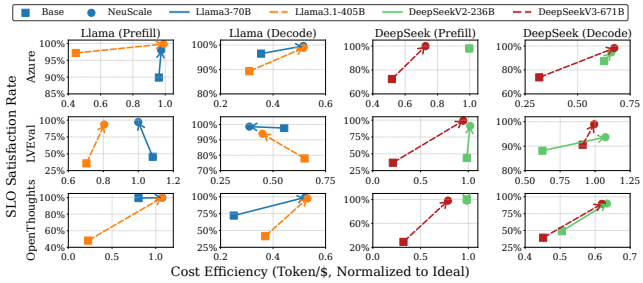


Figure 22: Monetary cost vs. SLO satisfaction rate. Different colors represent different models. Arrows indicate the improvement from *Base* to NeuScale.

3.61×), 1.43× (up to 5.64×), and 1.36× (up to 3.44×), as compared to *Base*. The trend across different models and traces is diverse, as they all have different SLO requirements for different requests. In general, NeuScale's benefits are more obvious for larger models and more diverse sequence lengths. This is because larger models require larger vPods, making the allocation search space larger. With more diverse sequence lengths, the drawback of *Base*'s one-size-fits-all allocation also becomes more obvious.

For most workloads, NeuScale improves SLO satisfaction rate over *Base*. Note that NeuScale prioritizes SLO guarantees, so it sometimes leads to lower energy/cost efficiency, as an SLO-compliant vPod configuration may not be the most efficient one (e.g., Llama3.1-405B in `OpenThoughts` in Figure 21). This is because *Base* applies a high pipeline parallelism degree to the largest sequence length, minimizing memory pressure (pipeline parallelism duplicates no model weights or activations, unlike tensor/expert parallelisms) and improving throughput. While deep pipelining improves efficiency, it adds significant latency overhead for small requests (longer ones are less affected, since their SLO latency target is higher). In contrast, NeuScale finds the best-fit vPod for each request to meet its SLO while improving efficiency with best effort.

In a few cases (e.g., DeepSeekV3-671B Prefill with `LVEval` in Figure 21), NeuScale's efficiency is slightly higher than *Ideal*, while the SLO satisfaction rate is lower than 100%. This implies some requests are executed on more efficient vPods than their best-fit vPods (i.e., when the best-fit vPod is coalesced or not allocated in the current epoch), while they suffer from SLO violations. The user can disable vPod group coalescing or adjust the auto-scaling epoch to trade off between SLO satisfaction and efficiency.

Impact of bursty workloads. We create synthetic traces with a 45-minute burst period, during which the workload bursts from 30 to 80 requests/minute, with average input/output sequence length increasing from 2K/128 to 100K/5K. As shown in Figure 23, when the request pattern shifts abruptly (at ~30 and 70 minutes), NeuScale suffers from ~10 minutes of SLO degradation. However, it quickly recovers from degradation and maintains its advantages over *Base*. Note that *Base*'s SLO satisfaction rate increases during the burst period, since the sequence lengths increase in the burst period,

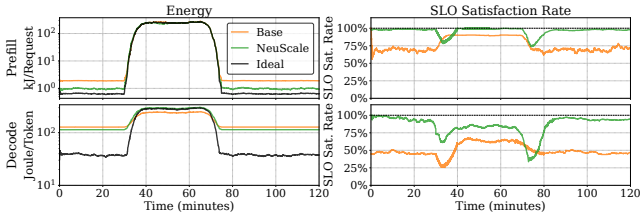


Figure 23: Energy efficiency and SLO satisfaction rate with a synthetic bursty trace.

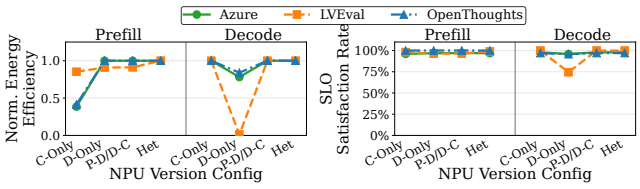


Figure 24: Energy efficiency (normalized to "Het") and SLO satisfaction rate of NeuScale with varying combinations of NPU versions. We study Llama3.1-405B as an example.

and *Base*'s vPods are provisioned for long sequence lengths, which cause SLO violations for some shorter sequences.

5.4 Sensitivity to Resource Availability

We now evaluate NeuScale with various mixes of NPU versions and resource quotas (i.e., limiting the number of chips).

Varying available NPU versions. We deploy NeuScale across three cluster configurations: (1) homogeneous clusters using exclusively NPU-C or NPU-D (C-Only or D-Only); (2) an "expert-choice" setup that maps prefill to NPU-D and decode to NPU-C (P-D/D-C in Figure 24); and (3) a heterogeneous cluster with both NPU-C and NPU-D available ("Het" in Figure 24). As shown in Figure 24, despite that the compute-optimized NPU-D "intuitively" matches the compute-intensive prefill phase, mapping all prefill requests to NPU-D degrades energy efficiency by 11% for the LVEval trace, as extremely long-context requests prefer NPU-C's larger memory capacity. Mapping decode requests to NPU-C performs well across all traces, aligning with our findings in §2.4. While most NPU versions can maintain high SLO satisfaction, D-Only struggles with LVEval because serving long-context requests requires many NPU-D chips, exacerbating ICI overhead. NeuScale alleviates the manual effort of NPU selection, automatically mapping requests to their best-fit hardware to improve efficiency and SLO satisfaction.

Varying resource quota. In Figure 25, we examine NeuScale under insufficient resource quota. We restrict the maximum number of NPU-C/D. To isolate the impact of fail-over allocation to older chips, we do not allow fail-over to any NPU-A/B. In general, provisioning enough vPod groups is essential for efficiency (to ensure each request is mapped to its best-fit vPod group), and having enough vPods per group is essential for SLO satisfaction (to prevent overloading). As we lower the resource quota, prefill phase suffers from up to 1.34× worse efficiency compared to unlimited resources. Decode is less affected, even though it demands more vPod groups

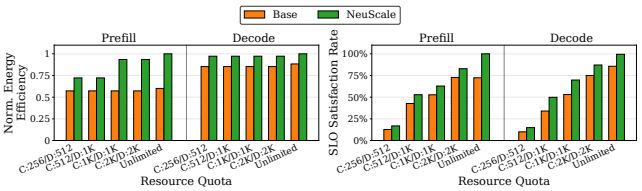


Figure 25: Energy efficiency (normalized to Unlimited) and SLO satisfaction rate with different resource quotas, on DeepSeekV3-671B with Azure as an example. "C:256/D:512" refers to the maximum number of NPU-C/NPU-D.

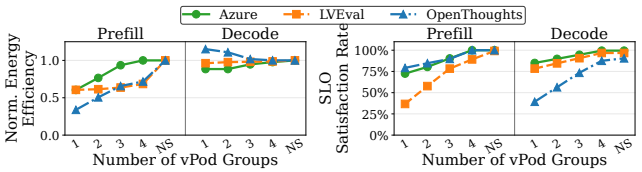


Figure 26: Energy efficiency (normalized to NS) and SLO satisfaction rate with different numbers of vPod groups. "1" is equivalent to *Base*. "NS" is NeuScale (no group number limit). We study DeepSeekV3-671B as an example.

than prefill (see Figure 20), because the decode energy efficiency of most requests is less sensitive to vPod configurations. Both prefill and decode suffer from low SLO satisfaction rate (as low as 14%/11%). While an insufficient resource quota inevitably degrades service quality, NeuScale still outperforms *Base* in all cases.

5.5 Benefits of Fine-Grained vPod Grouping

To better understand the benefits of having multiple vPod groups, we analyze how many vPod groups are required (i.e., the "granularity" of vPod grouping) to achieve the maximum efficiency with NeuScale. We create a coarse-grained multi-pool baseline by partitioning the sequence lengths evenly into a fixed number of ranges (e.g., the 0-33rd, 34th-66th, and 67th-100th percentiles) and creating a vPod group (pool) for each range. This represents an enhanced version of DynamoLLM [76]. The original DynamoLLM design relies on the user to select the number of pools and the vPod configuration for each pool, and it scales the number of vPods in each pool. In our experiments, we employ NeuScale's allocation algorithm to determine the optimal vPod configuration for each pool.

As shown in Figure 26, the benefit of having finer-grained vPod groups depends on the sequence length variance of the workload. Azure has a relatively narrow sequence length distribution, so 4 vPod groups are sufficient. LVEval and OpenThoughts exhibit massive variance in input and output lengths, respectively. Hence, they need more vPod groups for better efficiency and SLO satisfaction rate. For OpenThoughts, the energy efficiency is the best with one vPod group, because more requests are batched together. However, this also causes significant SLO violations due to interference between requests with diverse sequence lengths. Having more vPod groups helps improve performance isolation between these requests, which improves SLO satisfaction significantly. As

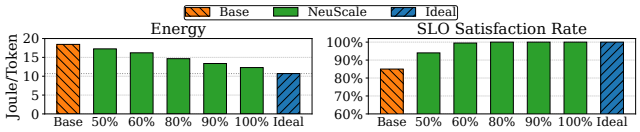


Figure 27: Energy efficiency and SLO satisfaction rate of decode under various output length prediction accuracies. Prefill is not affected. We show DeepSeekV3-671B with Azure trace as an example due to space limitations.

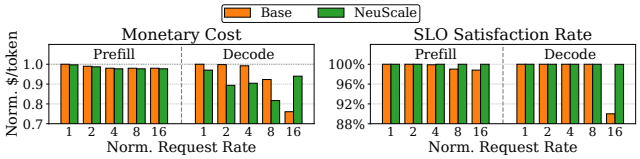


Figure 28: Monetary cost efficiency and SLO satisfaction rate with Llama3-8B on a real TPU cluster.

different workloads have diverse sequence length patterns, relying on a fixed number of pools is inherently sub-optimal. NeuScale overcomes this and can adapt to various workloads.

5.6 Impact of Output Length Prediction Accuracy

In Figure 27, we simulate different output sequence length prediction accuracies and analyze the impact on decode energy efficiency and SLO satisfaction rate. With 60% accuracy (the state-of-the-art predictors [63, 65, 80]), NeuScale already achieves significant energy savings over *Base*, and the accuracy is sufficient for near-perfect SLO satisfaction. The misprediction penalty on SLO is largely hidden, as NeuScale proactively re-predicts the output length and migrates requests to their best-fit vPod groups accordingly (see §3.4). Output length prediction is still an active research direction. As the predictor accuracy continues to improve in the future, the energy savings of NeuScale improve from 12% at 60% accuracy to up to 33% at perfect accuracy.

5.7 Evaluation on Real TPU Cluster

In Figure 28, we evaluate NeuScale on a small TPU cluster (see the setup in §4) using Llama3-8B and the Azure trace [76] (a production LLM serving trace as described in §5.1). As the cloud TPU stack does not expose a public API for measuring power, we report monetary cost. For this workload, TPUv6e is the most cost-efficient chip for both prefill and decode (see Figure 2), so we stress the system by sweeping request rates (from 60 requests/min to 16×) until TPUv6e capacity becomes insufficient. NeuScale maintains near-100% SLO satisfaction across all request rates by first using TPUv6e and then automatically failing over to TPUv5e and TPUv4 when TPUv6e chips run out. *Base* only auto-scales within the TPUv6e pool, causing significant SLO violations at high request rates.

The trends differ between prefill and decode. For prefill, throughput saturates even at small batch sizes, so increasing the request rate provides little additional batching benefit, and the per-token cost remains nearly flat. For decode, higher request rates enable

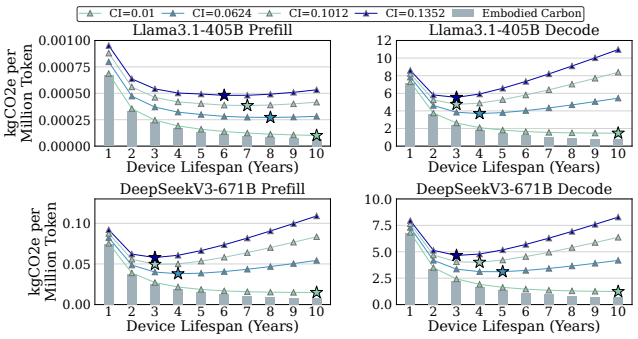


Figure 29: Lifetime carbon emission of different device lifespans over 10 years, assuming the energy efficiency improves every year by the ratio between NPU-C and NPU-A. CI is carbon intensity. The optimal lifetime with minimal carbon emission is highlighted with stars.

larger batches, reducing per-token cost for both NeuScale and *Base*, and NeuScale improves cost efficiency by up to 11.2% over *Base*. At 16× request rate, NeuScale incurs higher decode cost than *Base* because some requests are served on less efficient TPUv5e/TPUv4 fail-over vPods. This is a necessary tradeoff for preserving SLO under TPUv6e scarcity. This trend is consistent with the results in our simulation studies: by leveraging NPU heterogeneity, NeuScale always maintains or improves SLO satisfaction rate over *Base* while opportunistically improving cost efficiency.

5.8 Carbon Efficiency and Sustainability

The carbon footprint of an NPU chip consists of embodied carbon (emissions during manufacturing) and operational carbon (emissions due to runtime electricity consumption) [72]. NeuScale directly reduces the operational carbon by improving energy efficiency. It also provides an efficient way to reuse old NPU chips, thereby reducing their lifetime carbon footprint. We quantify the embodied carbon based on Google TPUs [72] and the operational carbon using the carbon intensity (carbon emission per unit energy consumed) reported by cloud providers [23]. For the "datacenter taxes" (e.g., cooling, power distribution, building maintenance), we assume a power usage effectiveness (PUE) factor of 1.09 [26]. We vary the carbon intensity to study the impact of deploying more renewable energy (which reduces carbon intensity).

Figure 29 quantifies the carbon footprint of LLM services, assuming various hardware update frequencies. CI=0.01 is a hypothetical future value. Other CIs are real values from a recent Google environmental report [26]. With a 1-year lifespan (no NPU reuse), 70.0%–98.7% of total emissions are embodied carbon. As we reuse NPUs, we can improve carbon efficiency by amortizing the embodied carbon. Beyond a certain lifespan, the carbon efficiency starts to drop. This is because the operational carbon becomes relatively larger, as older chips are less energy-efficient than newer chips. For our studied workloads, the carbon-optimal device lifespan is at least 3–6 years. Since new NPUs are deployed at a faster pace (e.g., every 1–2 years), reusing old NPUs to achieve their optimal lifespans is a promising way to improve datacenter carbon efficiency.

6 Discussion

Support for future NPU chip design. NeuScale provides an efficient way to integrate new NPU generations into existing NPU clusters. Based on our study, customizing NPU pod configurations for different workloads provides significant benefits. This reflects the trend towards building more specialized NPU chips, such as prefill chips and decode chips [10]. Such specialization introduces more heterogeneity in the NPU cluster. NeuScale can automatically map prefill and decode phases to the correct NPU versions.

Support for evolving ML workloads. Given any per-chip execution graph of an ML model, NeuScale can learn its arithmetic intensity and perform roofline-based analysis with the ML compiler. To support a new ML model architecture, the major effort is to implement the parallelism enumeration pass. This is acceptable, as the ML framework or compiler typically needs to be updated to support new parallelism strategies [46, 74, 98].

Support for heterogeneous accelerator architectures. As we build modern AI infrastructure, system architects should treat the heterogeneity of AI chips as a first-class citizen in the design. The system infrastructure should be workload-aware, and be able to map a workload to the best-fit chips. NeuScale focuses on heterogeneous NPUs, but its core idea can be applied to other accelerators. The vPod abstraction already employs a generic `vChipConfig` that can represent other AI chips (e.g., GPUs), and the `ICITopology` can be generalized to express different network topologies (e.g., all-to-all NVSwitch-connected GPUs). As cloud platforms typically deploy AI chips from different vendors, NeuScale paves the way for managing heterogeneous AI chips in a unified way.

7 Related Work

Heterogeneous cluster management. Prior studies mostly focused on heterogeneous CPUs/GPUs (see Table 1). For CPUs, they mapped heterogeneous workloads to homogeneous CPU cores by right-sizing the VM core count and memory size [17, 68, 88]. For GPUs, they investigated optimal parallelisms for ML models across heterogeneous GPUs [39, 51, 55, 84, 92] and heterogeneous GPU scheduling for ML training [50, 56, 57, 77, 95]. XSched [73] developed a scheduling framework for diverse accelerator types and enabled preemptive scheduling on a single chip. HeteCCL [37] optimizes network communications between heterogeneous GPUs for LLM training. NeuScale targets cluster scheduling and solves a completely different problem. For resource allocation, prior works primarily relied on heuristics or profiling to build performance models and find the optimal configuration (e.g., GPU version and clock frequency) [31, 76], which would incur high profiling overhead on NPUs. Unlike CPUs/GPUs, NPUs exhibit distinct compute capabilities and performance characteristics. NeuScale solves this challenge for NPUs using a lightweight yet accurate roofline model.

Auto-scaling frameworks. Auto-scaling for CPU workloads is well-supported by cloud platforms [5, 53, 70] and cluster orchestration frameworks [83] (see Table 1). They primarily focused on configuring the core count and memory size of each VM [17, 68, 88]. Some work leveraged reinforcement learning to make auto-scaling decisions for CPU workloads [64, 66]. None of them can directly work for NPUs. Most prior works on GPU focused on adjusting the number of homogeneous GPU instances [15, 24, 95]. NeuScale is the first auto-scaling framework for heterogeneous NPUs. Most recently, DynamoLLM [76] proposed a scheduling mechanism that organizes different GPU instance types (i.e., GPU count) into pools (similar to vPod groups) based on sequence lengths of incoming LLM requests. It is designed for homogeneous GPUs and relies on an expensive profiling-based allocation approach. Furthermore, it still relies on the user to configure the number of pools and the GPU instance configuration for each pool, and cannot automatically schedule across heterogeneous instance types.

LLM serving. Researchers have been optimizing LLM serving recently [3, 16, 35, 38, 43, 45, 48, 93]. For instance, prefill-decode disaggregation (PDD) [61, 99] was proposed to address the distinct resource demands between these phases. NeuScale maximizes the benefits of PDD by mapping the two phases to their best-fit vPods. Load balancing [11, 44, 78] for homogeneous LLM instances can be employed to optimize request scheduling within a vPod group. SpotServe[52] leverages spot instances to serve LLMs. NeuScale can support "spot vPods" by treating vPod preemptions as vPod failures. DynamoLLM [76] configures DVFS for different requests to optimize energy efficiency. As future work, NeuScale can be extended to include DVFS as a configurable parameter.

8 Conclusion

We develop NeuScale, a new auto-scaling framework for LLM serving on heterogeneous NPU chips. NeuScale improves energy/cost efficiency and SLO satisfaction for LLM services. It enables NPU reuse, facilitating its sustainable development and deployment.

References

[1] "vllm-project/router," accessed: 2026-06. [Online]. Available: https://github.com/vllm-project/router

[2] H. A. Abdelhafez, C. Zimmer, S. S. Vazhkudai, and M. Ripeanu, "Ahead: A tool for projecting next-generation hardware enhancements on gpu-accelerated systems," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 583–592.

[3] A. Agrawal, N. Kedia, A. Panwar, J. Mohan, N. Kwatra, B. Gulavani, A. Tumanov, and R. Ramjee, "Taming throughput-latency tradeoff in llm inference with sarathi-serve," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 117–134. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/agrawal

[4] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation," in *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 95–105. [Online]. Available: https://doi-org.proxy2.library.illinois.edu/10.1145/215399.215427

[5] Amazon Web Services, Inc., "Aws auto scaling," nov 2025. [Online]. Available: https://aws.amazon.com/autoscaling/

[6] AMD, "Axi high bandwidth memory controller logicore ip product guide (pg276)." [Online]. Available: https://docs.amd.com/r/en-US/pg276-axi-hbm

[7] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, G. Chauhan, A. Chourdia, W. Constable, A. Desmaison, Z. DeVito, E. Ellison, W. Feng, J. Gong, M. Gschwind, B. Hirsh, S. Huang, K. Kalambarkar, L. Kirsch, M. Lazos, M. Lezcano, Y. Liang, J. Liang, Y. Lu, C. K. Luk, B. Maher, Y. Pan, C. Puhrsch, M. Reso, M. Saroufim, M. Y. Siraichi, H. Suk, S. Zhang, M. Suo, P. Tillet, X. Zhao, E. Wang, K. Zhou, R. Zou, X. Wang, A. Mathews, W. Wen, G. Chanan, P. Wu, and S. Chintala, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 929–947.

[8] Anthropic, "Claude," https://claude.ai, 2024.

[9] AWS, "Amazon ec2 trn1/trn1n architecture," nov 2025. [Online]. Available: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/about-neuron/arch/neuron-hardware/trn1-arch.html

[10] J. Chen, Y. Zhuang, and H. Zhang, "Disaggregated inference: 18 months later," nov 2025. [Online]. Available: https://hao-ai-lab.github.io/blogs/distserve-retro/

[11] K. Cheng, W. Hu, Z. Wang, H. Peng, J. Li, and S. Zhang, "Slice-level scheduling for high throughput and load balanced llm serving," *arXiv preprint arXiv:2406.13511*, 2024.

[12] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S002626921630026X

[13] DeepSeek-AI, "Deepseek-v3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2412.19437

[14] A. Dhamba and A. V. Kulkarni, "Design considerations for high bandwidth memory controller." [Online]. Available: https://www.design-reuse.com/articles/41186/design-considerations-for-high-bandwidth-memory-controller.html

[15] A. Elmeleegy, H. Kim, H. Zhou, I. Dhanani, M. Khadkevich, O. Kahalon, and V. S. Mailthody, "Nvidia dynamo adds gpu autoscaling, kubernetes automation, and networking optimizations," may 2025. [Online]. Available: https://developer.nvidia.com/blog/nvidia-dynamo-adds-gpu-autoscaling-kubernetes-automation-and-networking-optimizations/

[16] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *Journal of Machine Learning Research*, vol. 23, no. 120, pp. 1–39, 2022.

[17] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 195–204.

[18] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, "{ServerlessLLM}:{Low-Latency} serverless inference for large language models," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.

[19] GitHub, "Github copilot," https://github.com/features/copilot, 2024.

[20] Google, "Jetstream." [Online]. Available: https://github.com/AI-Hypercomputer/JetStream

[21] ——, "System architecture - cloud TPU," 2022. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[22] ——, "XLA: Optimizing Compiler for Machine Learning," 2023. [Online]. Available: https://www.tensorflow.org/xla

[23] ——, "2024 environmental report," 2024. [Online]. Available: https://www.gstatic.com/gumdrop/sustainability/google-2024-environmental-report.pdf

[24] ——, "Configure autoscaling for llm workloads on gpus with google kubernetes engine (gke)," nov 2025. [Online]. Available: https://docs.cloud.google.com/kubernetes-engine/docs/how-to/machine-learning/inference/autoscaling

[25] ——, "Configure autoscaling for llm workloads on tpus," aug 2025. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/how-to/machine-learning/inference/autoscaling-tpu

[26] Google, "Google 2025 environmental report," https://sustainability.google/reports/google-2025-environmental-report/, 2025, accessed: 2026-02-28.

[27] Google, "Mlops: Continuous delivery and automation pipelines in machine learning," nov 2025. [Online]. Available: https://docs.cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning

[28] ——, "Vertex AI platform," may 2025. [Online]. Available: https://cloud.google.com/vertex-ai

[29] ——, "Cloud TPU pricing," https://cloud.google.com/tpu/pricing?hl=en, 2026.

[30] Google DeepMind, "Gemini," https://gemini.google.com, 2024.

[31] T. Griggs, X. Liu, J. Yu, D. Kim, W.-L. Chiang, A. Cheung, and I. Stoica, "Mélange: Cost efficient large language model serving by exploiting gpu heterogeneity," 2024. [Online]. Available: https://arxiv.org/abs/2404.14527

[32] J. Gu, P. Wang, I. D. N. Araya, K. Huang, and M. Gerndt, "Has-gpu: Efficient hybrid auto-scaling with fine-grained gpu allocation for slo-aware serverless inferences," 2025. [Online]. Available: https://arxiv.org/abs/2505.01968

[33] E. Guha, R. Marten, S. Keh, N. Raoof, G. Smyrnis, H. Bansal, M. Nezhurina, J. Mercat, T. Vu, Z. Sprague, A. Suvarna, B. Feuer, L. Chen, Z. Khan, E. Frankel, S. Grover, C. Choi, N. Muennighoff, S. Su, W. Zhao, J. Yang, S. Pimpalgaonkar, K. Sharma, C. C.-J. Ji, Y. Deng, S. Pratt, V. Ramanujan, J. Saad-Falcon, J. Li, A. Dave, A. Albalak, K. Arora, B. Wulfe, C. Hegde, G. Durrett, S. Oh, M. Bansal, S. Gabriel, A. Grover, K.-W. Chang, V. Shankar, A. Gokaslan, M. A. Merrill, T. Hashimoto, Y. Choi, J. Jitsev, R. Heckel, M. Sathiamoorthy, A. G. Dimakis, and L. Schmidt, "Openthoughts: Data recipes for reasoning models," 2025. [Online]. Available: https://arxiv.org/abs/2506.04178

[34] L. Gwennap, "Tenstorrent scales ai performance: New multicore architecture leads in data-center power efficiency," 2020. [Online]. Available: https://www.linleygroup.com/mpr/article.php?id=12287

[35] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons, "Pipedream: Fast and efficient pipeline parallel dnn training," *arXiv preprint arXiv:1806.03377*, 2018.

[36] S. He, W. Cai, J. Huang, and A. Li, "Capacity-aware inference: Mitigating the straggler effect in mixture of experts," 2026. [Online]. Available: https://arxiv.org/abs/2503.05066

[37] C. Hei, J. Li, J. Cao, C. Gao, X. Sha, T. Liu, D. Zhang, E. Zhai, and X. Wang, "HeteCCL: Synthesizing Near-Optimal collective communication schedules for heterogeneous GPU clusters," in *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. Renton, WA: USENIX Association, May 2026, pp. 2533–2551. [Online]. Available: https://www.usenix.org/conference/nsdi26/presentation/hei

[38] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *Advances in neural information processing systems*, vol. 32, 2019.

[39] X. Jia, L. Jiang, A. Wang, W. Xiao, Z. Shi, J. Zhang, X. Li, L. Chen, Y. Li, Z. Zheng, X. Liu, and W. Lin, "Whale: Efficient giant model training over heterogeneous GPUs," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 673–688. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/jia-xianyan

[40] N. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. A. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*, Orlando, FL, USA, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589350

[41] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*, Virtual Event, 2021.

[42] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, June 2020.

[43] V. A. Korthikanti, J. Casper, S. Lym, L. McAfee, M. Andersch, M. Shoeybi, and B. Catanzaro, "Reducing activation recomputation in large transformer models," *Proceedings of Machine Learning and Systems*, vol. 5, pp. 341–353, 2023.

[44] F. Kossmann, B. Fontaine, D. Khudia, M. Cafarella, and S. Madden, "Is the gpu half-empty or half-full? practical scheduling techniques for llms," *arXiv preprint arXiv:2410.17840*, 2024.

[45] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient memory management for large language model serving with pagedattention," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 611–626.

[46] D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen, "Gshard: Scaling giant models with conditional computation and automatic sharding," 2020. [Online]. Available: https://arxiv.org/abs/2006.16668

[47] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," 2022. [Online]. Available: https://arxiv.org/abs/2105.13120

[48] B. Lin, C. Zhang, T. Peng, H. Zhao, W. Xiao, M. Sun, A. Liu, Z. Zhang, L. Li, X. Qiu, S. Li, Z. Ji, T. Xie, Y. Li, and W. Lin, "Infinite-llm: Efficient llm service for long context with distattention and distributed kvcache," *arXiv preprint arXiv:2401.02669*, 2024.

[49] Llama Team, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[50] J. Ma, G. Zuo, K. Loughlin, X. Cheng, Y. Liu, A. M. Eneyew, Z. Qi, and B. Kasikci, "A hypervisor for shared-memory fpga platforms," in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[51] Y. Mei, Y. Zhuang, X. Miao, J. Yang, Z. Jia, and R. Vinayak, "Helix: Serving large language models over heterogeneous gpus and network via max-flow," ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 586–602.

[52] X. Miao, C. Shi, J. Duan, X. Xi, D. Lin, B. Cui, and Z. Jia, "Spotserve: Serving generative large language models on preemptible instances," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1112–1127. [Online]. Available: https://doi.org/10.1145/3620665.3640411

[53] Microsoft, "Autoscale in azure monitor," nov 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-overview

[54] Microsoft Azure, "Azure public dataset: Azure LLM inference trace 2023," GitHub, 2023, accessed: 2026-06-17. [Online]. Available: https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md

[55] Z. Mo, J. Liao, H. Xu, Z. Zhou, and C. Xu, "Hetis: Serving llms in heterogeneous gpu clusters with fine-grained and dynamic parallelism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '25. ACM, Nov. 2025, p. 1710–1724. [Online]. Available: http://dx.doi.org/10.1145/3712285.3759784

[56] Z. Mo, H. Xu, and C. Xu, "Heet: Accelerating elastic training in heterogeneous deep learning clusters," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 499–513.

[57] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia, "Heterogeneity-Aware cluster scheduling policies for deep learning workloads," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020, pp. 481–498. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

[58] OpenAI, "Chatgpt," https://chat.openai.com, 2024.

[59] OpenXLA Authors, "Hlo passes," nov 2025. [Online]. Available: https://openxla.org/xla/hlo_passes#analysis_passes

[60] ——, "Xla tooling," nov 2025. [Online]. Available: https://openxla.org/xla/tools

[61] P. Patel, E. Choukse, C. Zhang, A. Shah, Í. Goiri, S. Maleki, and R. Bianchini, "Splitwise: Efficient generative llm inference using phase splitting," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 118–132.

[62] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters," in *Proceedings of the 13th European Conference on Computer Systems (EuroSys'18)*, Porto, Portugal, Apr. 2018.

[63] G. Piotrowski, M. Bystroński, M. Hołysz, J. Binkowski, G. Chodak, and T. J. Kajdanowicz, "When will the tokens end? graph-based forecasting for LLMs output length," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 4: Student Research Workshop)*, J. Zhao, M. Wang, and Z. Liu, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 843–848. [Online]. Available: https://aclanthology.org/2025.acl-srw.61/

[64] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 805–825. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/qiu

[65] H. Qiu, W. Mao, A. Patke, S. Cui, S. Jha, C. Wang, H. Franke, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "Efficient interactive llm serving with proxy model-based sequence length prediction," 2024. [Online]. Available: https://arxiv.org/abs/2404.08509

[66] H. Qiu, W. Mao, C. Wang, H. Franke, A. Youssef, Z. T. Kalbarczyk, T. Başar, and R. K. Iyer, "AWARE: Automate workload autoscaling with reinforcement learning in production cloud systems," in *Proccedings of 2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, Jul. 2023, pp. 387–402. [Online]. Available: https://www.usenix.org/conference/atc23/presentation/qiu-haoran

[67] Rambus, "Hbm3e / hbm3 controller ip." [Online]. Available: https://www.rambus.com/interface-ip/hbm/hbm3-controller/

[68] J. Rao, X. Bu, C.-Z. Xu, and K. Wang, "A distributed self-learning approach for elastic provisioning of virtualized cloud resources," in *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2011, pp. 45–54.

[69] RUN:AI, "Google TPU Architecture and Performance Best Practices," 2022. [Online]. Available: https://www.run.ai/guides/cloud-deep-learning/google-tpu

[70] K. Rzadca, P. Findeisen, J. Swiderski, P. Zych, P. Broniek, J. Kusmierek, P. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: workload autoscaling at google," in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*, Heraklion, Greece, 2020.

[71] S. M. Sajal, T. Zhu, B. Urgaonkar, and S. Sen, "Traceupscaler: Upscaling traces to evaluate systems at high load," in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 942–961.

[72] I. Schneider, H. Xu, S. Benecke, D. Patterson, K. Huang, P. Ranganathan, and C. Elsworth, "Life-cycle emissions of ai hardware: A cradle-to-grave approach and generational trends," 2025. [Online]. Available: https://arxiv.org/abs/2502.01671

[73] W. Shen, M. Han, J. Liu, R. Chen, and H. Chen, "XSched: Preemptive scheduling for diverse XPUs," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. Boston, MA: USENIX Association, Jul. 2025, pp. 671–692. [Online]. Available: https://www.usenix.org/conference/osdi25/presentation/shen-weihang

[74] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-lm: Training multi-billion parameter language models using model parallelism," 2020. [Online]. Available: https://arxiv.org/abs/1909.08053

[75] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Sigcomm '15*, 2015.

[76] J. Stojkovic, C. Zhang, I. Goiri, J. Torrellas, and E. Choukse, " DynamoLLM: Designing LLM Inference Clusters for Performance and Energy Efficiency ," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2025, pp. 1348–1362. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA61900.2025.00102

[77] F. Strati, Z. Zhang, G. Manos, I. S. Périz, Q. Hu, T. Chen, B. Buzcu, S. Han, P. Delgado, and A. Klimovic, "Sailor: Automating distributed training over dynamic, heterogeneous, and geo-distributed clusters," in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, ser. SOSP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 204–220.

[78] B. Sun, Z. Huang, H. Zhao, W. Xiao, X. Zhang, Y. Li, and W. Lin, "Llumnix: Dynamic scheduling for large language model serving," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 173–191. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/sun-biao

[79] T. Tang, S. Li, L. Nai, N. Jouppi, and Y. Xie, "Neurometer: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 841–853.

[80] Y. Tao, Y. Zhang, M. T. Dearing, X. Wang, Y. Fan, and Z. Lan, "Prompt-aware scheduling for low-latency llm serving," 2025. [Online]. Available: https://arxiv.org/abs/2510.03243

[81] The JAX Authors, "Jax: High performance array computing." [Online]. Available: https://docs.jax.dev/en/latest/

[82] The Kubernetes Authors, "Kubernetes scheduler," 2023. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/

[83] ——, "Horizontal pod autoscaling," 2025. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[84] T. Um, B. Oh, M. Kang, W.-Y. Lee, G. Kim, D. Kim, Y. Kim, M. Muzzammil, and M. Jeon, "Metis: Fast automatic distributed training on heterogeneous GPUs," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 563–578.

[85] T. Underwood, "SRE for ML: The first 10 years and the next 10." USENIX Association, Oct. 2021.

[86] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[87] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[88] L. Wei, C. H. Foh, B. He, and J. Cai, "Towards efficient resource allocation for heterogeneous workloads in iaas clouds," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 264–275, 2018.

[89] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, apr 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[90] J. Xie, Z. Chen, R. Zhang, X. Wan, and G. Li, "Large multimodal agents: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2402.15116

[91] Y. Xue, Y. Liu, L. Nai, and J. Huang, "V10: Hardware-assisted npu multi-tenancy for improved resource utilization and fairness," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA'23)*, Orlando, FL, USA, 2023.

[92] X. Yi, S. Zhang, Z. Luo, G. Long, L. Diao, C. Wu, Z. Zheng, J. Yang, and W. Lin, "Optimizing distributed training deployment in heterogeneous gpu clusters," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, 2020, pp. 93–107.

[93] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Kim, and B.-G. Chun, "Orca: A distributed serving system for Transformer-Based generative models," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 521–538. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/yu

[94] T. Yuan, X. Ning, D. Zhou, Z. Yang, S. Li, M. Zhuang, Z. Tan, Z. Yao, D. Lin, B. Li, G. Dai, S. Yan, and Y. Wang, "Lv-eval: A balanced long-context benchmark with 5 length levels up to 256k," 2024.

[95] G. Yun, J. Kang, H. Jeong, S. Eom, M. Jang, and Y.-r. Choi, "Jabas: Joint adaptive batching and automatic scaling for dnn training on heterogeneous gpus," in *Proceedings of the Twentieth European Conference on Computer Systems*, ser. EuroSys '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 769–786.

[96] D. Zhang, H. Wang, Y. Liu, X. Wei, Y. Shan, R. Chen, and H. Chen, "Blitzscale: Fast and live large model autoscaling with o(1) host caching," 2025. [Online]. Available: https://arxiv.org/abs/2412.17246

[97] H. Zhang, A. Ning, R. B. Prabhakar, and D. Wentzlaff, "Llmcompass: Enabling efficient hardware design for large language model inference," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, 2024.

[98] L. Zheng, Z. Li, H. Zhang, Y. Zhuang, Z. Chen, Y. Huang, Y. Wang, Y. Xu, D. Zhuo, E. P. Xing, J. E. Gonzalez, and I. Stoica, "Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning," in *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, Jul. 2022.

[99] Y. Zhong, S. Liu, J. Chen, J. Hu, Y. Zhu, X. Liu, X. Jin, and H. Zhang, "Distserve: disaggregating prefill and decoding for goodput-optimized large language model serving," in *Proceedings of the 18th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'24. USA: USENIX Association, 2025.

[100] Y. Zu, A. Ghaffarkhah, H.-V. Dang, B. Towles, S. Hand, S. Huda, A. Bello, A. Kolbasov, A. Rezaei, D. Du, S. Lacy, H. Wang, A. Wisner, C. Lewis, and H. Bahini, "Resiliency at scale: Managing Google's TPUv4 machine learning supercomputer," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 761–774. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/zu